SYSTEM TEST METHODOLOGY

George Wynn Montgomery

THESIS

SYSTEM TEST METHODOLOGY

by

George Wynn Montgomery

June 1975

Thesis Advisor: N. F. Schneidewind

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): System Test Methodology	5. TYPE OF REPORT & PERIOD COVERED: Master's thesis; June 1975
	6. PERFORMING ORG. REPORT NUMBER: -
7. AUTHOR(s): George Wynn Montgomery	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: June 1975 / 13. NUMBER OF PAGES: 57
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Postgraduate School, Monterey, California 93940	15. SECURITY CLASS. (of this report): Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

software testing
system test
system development

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A large proportion of the cost of developing information processing systems is devoted to the testing phase of system development. At present the results of this large expenditure have been inadequate to suit the high reliability demanded of today's computer systems.

This research develops a framework of a system test methodology that can be used to provide a rational approach

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

20.

to the design of a system. The methodology uses a model that provides a modularized functional representation of a processing system. The model is used to investigate a broad class of system test problems. A simulation of the model was constructed. The use of simulation during system testing was discussed.

System Test Methodology

by

George Wynn Montgomery
Lieutenant, United States Navy
B.S., United States Naval Academy, 1966

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1975

ABSTRACT

A large proportion of the cost of developing information processing systems is devoted to the testing phase of system development. At present the results of this large expenditure have been inadequate to suit the high reliability demanded of today's computer systems.

This research develops a framework of a system test methodology that can be used to provide a rational approach to the design of a system. The methodology uses a model that provides a modularized functional representation of a processing system. The model is used to investigate a broad class of system test problems. A simulation of the model was constructed. The use of simulation during system testing was discussed.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

I. INTRODUCTION

A. MOTIVATION FOR SYSTEM TEST METHODOLOGY

Software is the major expense in computer systems today. As an example, the Air Force allocated between one billion dollars and one and a half billion dollars in 1972 for software development. This was about three times the annual expenditure on computer hardware and accounted for four to five percent of the Air Force budget for the year. Boehm [14, 15] indicates that these high figures are representative of the industry as a whole. He predicts that by 1985 software expenditures in the Air Force will account for ninety percent of the total ADP system costs. Of this enormous amount of money spent on software, a disproportionately large share was spent on testing and the trend is not one of improvement. Boehm states that "during the 1970s the Air Force can expect to spend almost half of its software budget for military space operations on the checkout and test phases of computer program implementation: two to three times as much as it will pay for having the program coded." With such an effort going into testing software, it should be relatively error free but this has not been the case historically. The Apollo Manned Spaceflight Program had one of the most tested systems in the world, yet major software failures occured in Apollos 8, 11, and 14. The failure on Apollo 11 occurred in the extremely critical phase of lunar landing. Things are no better on the civilian side, each new release of OS/360 has

around 1000 new software errors. It is not necessary to look at such large complicated systems to discover that present testing is inadequate. The person who has not had an encounter with a computer program error such as an incorrect billing is an unusual person in today's society. Since testing consumes such a large proportion of the resources allocated to system development and has produced such poor results, it is time to develop a new approach to system test.

B. TESTING PROBLEMS

1. Multiplicity of Testing Activities

Many of the terms used in the area of testing are subject to a wide variety of interpretations. The word "testing" has been misused and many non-testing activities have been associated with the word. Testing may be defined to be the process of determining if a system meets the stated functional specifications. Quite often debugging is thought of as a testing activity. This is incorrect. Debugging starts with a known error and works for a correction [9]. Recently, a significant body of literature and activity have been addressed to designing computer programs in a structured fashion in order to eliminate or minimize the occurrence of software errors [5, 10]. The theme of some of these efforts is that if we design programs correctly through structured programming, there will be very little need for testing. Although these efforts do a lot to reduce the potential for errors, they do not act as a substitute for testing.

Other testing activities include verification,

validation, certification, proof of correctness, and performance testing. Hetzel [9] discusses these activities in relation to program testing. Verification is concerned with the program's logical correctness based on execution of the program in a test environment. Validation is concerned with the logical correctness of a program in a given external environment. Certification implies an authoritative endorsement that a program is of a certain quality. A proof of correctness deals with the logical correctness without regard to the environment. Performance testing concentrates on the non-logical properties such as resource utilization. Each of these activities has much to offer. The problem arises when one of the approaches is assumed to equate to complete testing. It is clear that improved software quality must be approached from several fronts: improved design techniques, improved programming management, and improved testing methodology; rather than assuming that a panacea is present in a single approach.

2. Test Design

There are many fundamental questions that must be answered in designing a test of an information processing system. One such question is what should be tested? Too often a tester ends up testing an incomplete or modified version of the system that is easier to test than the real system. Often the tester is faced with an infinite set of input combinations to be tested. In this case, the question becomes how can a subset of the test inputs best be selected to throughly test the system? Another important issue is how should the test efforts be organized? It is important to get the most information about the system out of every test run. It is important to establish test data recording procedures at this time in order to insure that all error information will be recorded. This can be accomplished by

properly organizing the tests in a logical sequence. Tests should be related to types and sources of errors. Gruenberger [8] states that "part of the art of testing is knowing when to stop testing." This exposes a two sided question the test designer must face: when is the test finished and what can be said about the system when testing is stopped?

All these questions are futher compounded by the fact that there can be no set rule. Every system requires an original test procedure designed to fit its special requirements. Gruenberger suggests "that the intellectual effort to test a program is of the same order as that which created it." Perhaps if he had said properly tested, using a minimum of resources, the effort would be on the order of the original effort squared.

This thesis presents a test methodology that will help answer these questions. A model is presented that will serve as a framework for the construction of a logical approach to system testing.

C. A MODULAR APPROACH TO SYSTEM TESTING

A modular approach to system testing offers many advantages for the design of the test and the development of the system. The modular design involves breaking a large system into many small parts called modules. The intra-module functions are independent; however modules interact by means of standard interfaces. Each module performs a major function of the system.

Modularity improves system design. Modularity allows software to be portable. To an extent, modules may be

transferred among machines and operating systems. With standardization of modules, they may be shared among many applications. With modules being shared in this manner, the programming effort is reduced and the reliability of the program is increased since the modules will be tested with each application. The progam may be expanded easier and changes are easier to incorporate since the effect of a change is localized.

The testability of a modular system is greatly improved. Testing of different modules may be carried out in parallel. Standardization of modules yields a set of assertions that may be used as test criteria for the modules. Modules may be compiled separately and can be stored in a program library and accessed independently. Modularity allows testing a system early in the construction phase. Each module may be tested as soon as it has been constructed instead of waiting for the whole system to be completed before starting to test. Since modules can be reused, it will be economically feasible to do more testing.

A modular system was chosen for the model in order to take advantage of these improvements in system design and system test.

II. MODEL DESCRIPTION

A. THE FUNCTIONAL MODULE CONCEPT

1. Module Definition

In the representation of a system using the functional model, the lowest element of the system that was considered was the module. Since the word module has had wide use throughout the computer industry, it is necessary to completely define the desired application of the word in the model. A module is an entity that performs a function within the system. A function is an activity performed by the system such as a fast Fourier transform. The physical embodiment of a module is the wiring and circuit board of hardware and the source or object programs recorded on punched cards or magnetic tape or resident in memory, for computer software. By addressing modules by the function they perform, a module is freed from the distinction of being just hardware or just software. The proper modules to test a system is determined by the accessibility of information within a system. In order that a module be useful for testing purposes, the module must have accessible and controllable inputs and measurable outputs. Therefore, the modules would be chosen at the lowest level that the tester could ensure these properties.

A module receives inputs and transmits outputs

across a boundary. A boundary consists of a location within the system at which the inputs to a module or the outputs from a module may be measured. In order for the tester to access these inputs or outputs the boundary must be identifiable. In order to accomodate this requirement for an identiable boundary, it is necessary to consider the composition of modules. The composition of two modules would be a module performing the same functions as the original two modules. An example of composition of modules could involve two modules. One is a fast Fourier transform module and the other is a digital filter module. If it is impossible to identify a point to measure the output from the filter module to the Fourier transform module, the two could be considered as one module that performs the function of the filter and of the transform module. Thus, the system as a whole could be viewed as a module or a module could be considered to be a small unit of code. The proper level to identify modules will be indicated by the access to the flow of resources in the system and the functions performed by the system.

A module will be assumed to be free of internal errors for system test purposes. Internal errors are those types of errors which are found during unit testing of the module. This assumption is predicated on the fact that all modules will receive extensive individual unit testing before the system is assembled. If an error still exists within a module, the test system will detect it only as the error applies to the module's relationship with elements of the system as a whole. Assuming that the test plan is sufficent to detect all errors external to a module, the only way an error could go undetected would be if its actions were confined to the module itself.

The system may now be described as a collection of modules which has external inputs and external outputs. The

selection of modules must be such that every portion of the entire system is represented by a module and no portion is represented by more than one module.

In performing this function, the module utilizes resources provided to the system. These resources may be in the form of data, control signals, or physical resources including both hardware and software units. Thus a resource is an element of the system that is used by modules in performing a function of the system. Resources have two types of attributes. One type deals with the usage of the resource which is the amount or size of the resource that is assigned or available to be assigned. The other type of attribute of a resource deals with the contents of a particular resource such as the contents of a memory location or the actual value of a particular control signal. Resources have states. These states indicate the status of the resource. Some examples of the state of a resource are reading, writing, idle, file empty, file half full, or memory region assigned.

2. Task Definition

The work to be performed by a module may be represented as an ordered or random series of tasks. Tasks are the sub-functions performed by a module. A sub-function consists of the execution of a step in the algorithm which the module must execute in order to carry out its function. Thus, a module is the representation of a function within the system and tasks are the representation of the execution of the function. Examples of tasks are the computation of a simple function, storing the result in memory and outputting the result to the printer. This usage of the word task is syononymous with the use of the word "process" as it is used throughout the operating system literature. The precedence

of tasks is determined by the algorithm. It is also possible to have no precedence constraints. In this case any task could be executed whenever the resources were available.

In order to execute a task, the module goes through a series of states. The state of a module is the status of the module at a given time. A partial list of states that a module can enter includes: compute, wait for memory, wait for input/output, wait for cpu, idle, input processing, wait for another module to complete a task, wait for a resource, and interrupted state. The particular state of a module is a function of the set of inputs to the module, resource states, and its previous state. The outputs of a module are a function only of the state of the module. A primary state is a state that a module is required to enter in order to perform a task. Primary tasks include compute, input processing and output processing. A secondary state is a state in which the module accomplishes no work. Examples of secondary states would be the blocked state, wait for input or wait for cpu.

The system state is the set of module states. The system state changes when any module changes state.

3. Model Notation

The following is a list of symbols used to describe the model. Each symbol is followed by the definition of that symbol as it is used in this system of notation.

* i ----- Module designation,

* j_i ----- Current state of module i,

* k_i ----- Next state of module i,

* I_{ijt} ----- Vector of inputs at module i when module is in state j and input starts at time t,

* $O_{ikt'}$ ----- Vector of outputs from module i after the module has transitioned to state k and output starts at time t',

* T_{ijk} ----- Time at which transition of module i from state j to state k occurs,

* dT_{ij} ----- Amount of time which module i spends in state j,

* R_{ij} ----- Set of resources used by module i when in state j,

* $(l_1, l_2, \ldots, l_n)_{ij}$ ----- State of n resources when module i is in state j,

* $(t_1, t_2, \ldots, t_n)_{ij}$ ----- Time which module i uses n resources when in state j.

4. Model as a Directed Graph

It is possible to represent the system as a series of directed graphs. One graph would be required for each module. The nodes of the graph would represent module states and the arcs would represent state transitions. Other information could be portrayed on the graph. The state

dependent information could be associated with the node. This would include the current state of the module, the set of resources used by the module in that state, the state vector for the resources used by the module, the vector of inputs to the module, the vector of outputs from the module and the amount of time the module spends in the state. The arcs could be labelled with the time that the module requires to transition from the source state to the destination state as is shown in Fig 1. In this figure, the module i transitions from state j to state k at time T_{ijk}.

These directed graphs would give the tester a convenient means of visually representing the activity of the module. The tester might prefer to show only the primary states of the module and the idle state instead of showing all possible states of the system.

5. Time Domain of a Module

A property of a module is that it uses the resources of the system only at certain times. One of the major problems of testing computer systems is trying to identify when two or more modules will be competing for the same resource or resources. The problem is further compounded by the system that uses multiple CPU's which are running asynchronously. The concept of time domain will be useful to address this problem area. A time domain of a module consists of the times that resources are in use. A graph of the time domains of the modules of the system would be a useful abstraction of the system for the analysis of the timing problem. The resources of the system could be represented on the vertical axis with time expanding along the horizontal axis from the orgin. Each area so represented

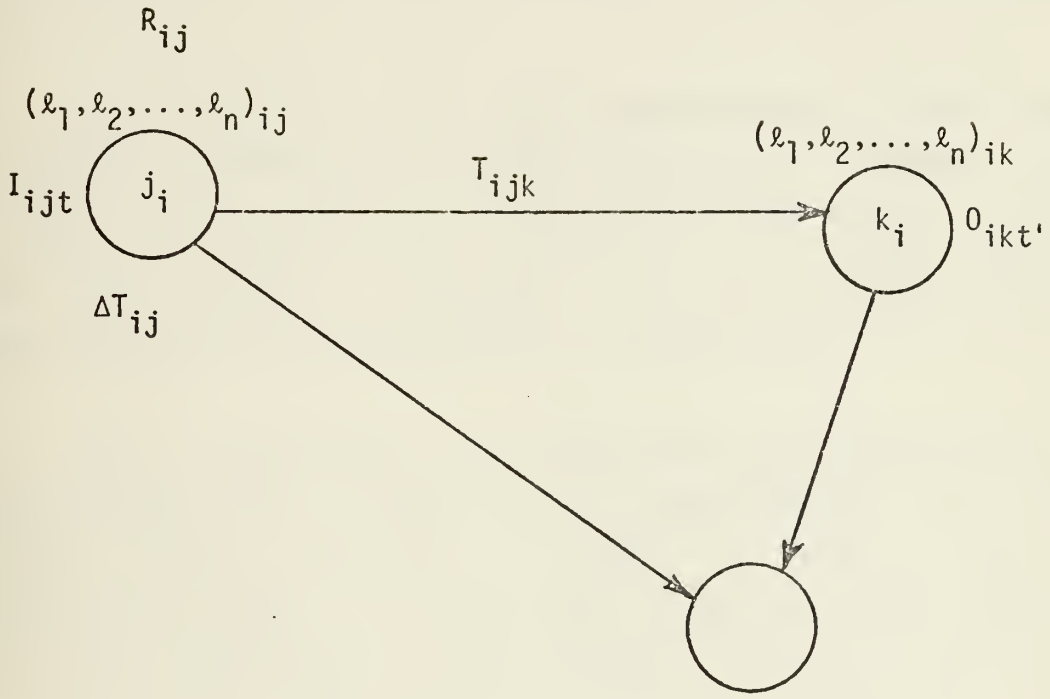


Fig 1 DIRECTED GRAPH OF MODULE STATE TRANSITION

should be labelled with the module and the amount of the resource required. The time domain of a module would be the summation of the area representing the resources used by the module at any given time. Any intersection of time domains would represent a potential error only if the total demands of the modules exceed the maximum resource available. It is also possible for one module to exceed the maximum resources available.

One problem with this representation of the system is finding a timing system that applies to all modules when the system has modules working asynchronously. In this case the time axis would be the elapsed time from some critical event that appears throughout the system. The changes in system state will be referenced to this event.

It is possible to consolidate the represention of module states into a system state representation and show resource usage conflicts in terms of system states as indicated in Fig 2. In this figure there are five types of resources available to the system. They are labelled R1 through R5. The amount of each resource is indicated on the vertical axis. For example, there are six units of R2 available. There are two resource conflicts protrayed in this system. One occurs in system state S_2. Here one module requires four units of resource R4 and another module requires two units of R4. The conflict occurs because there are only four units of R4 available. The conflict is denoted by a cross-hatched area. The other conflict is in system state S_4. A module has requested six units of resource R3 when only two units are available to the system [11].

The construction of such a graph would be infeasible

to do by hand for a real system. A program could be written to produce this type of graph from the time domains of the modules. On this graph the computer could identify resource usage conflicts. The program would be independent of the system being tested; therefore, it could be used on many different projects.

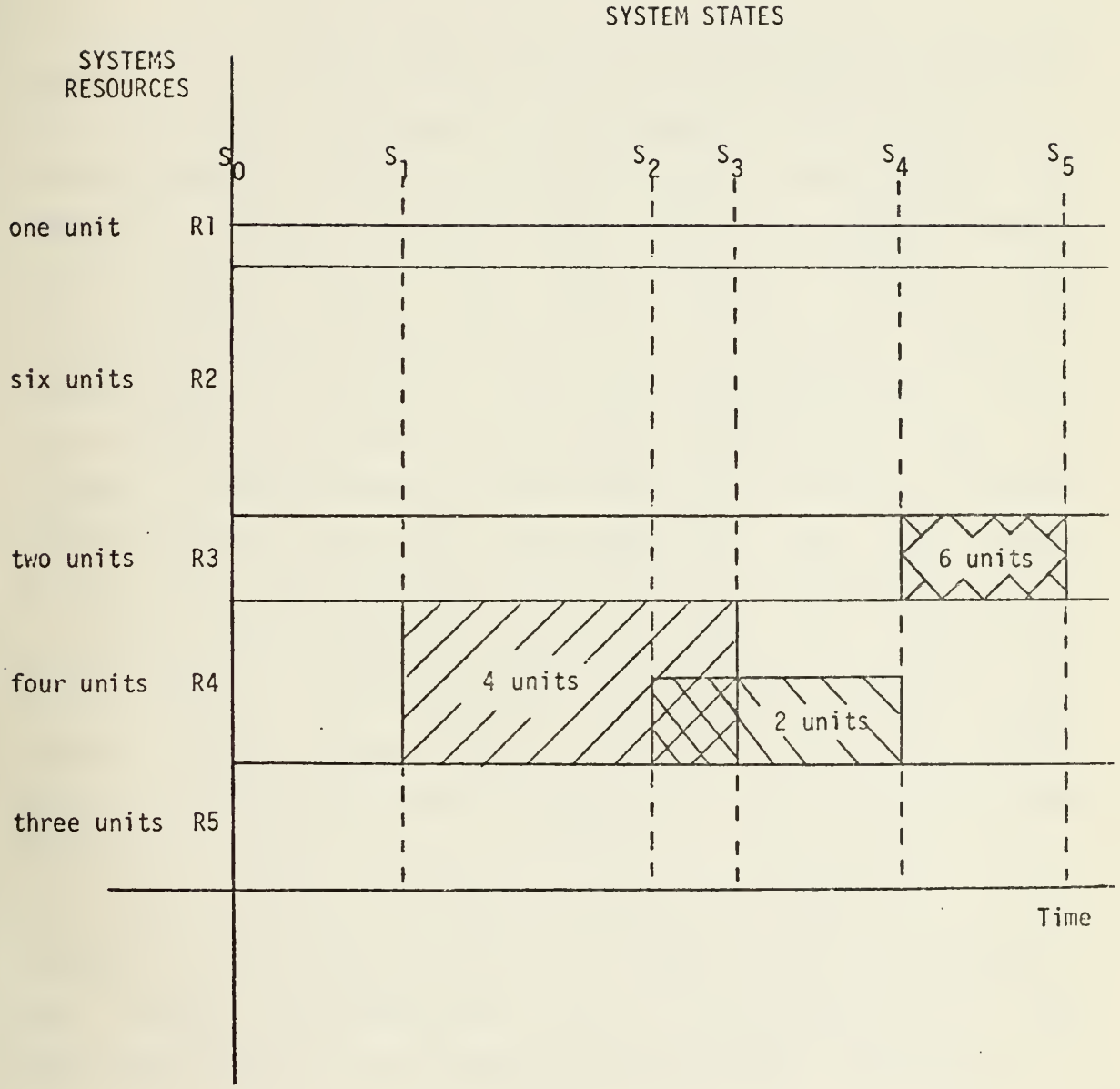


Fig 2 RESOURCE CONFLICTS VS SYSTEM STATE

B. APPLICATION OF MODEL TO TESTING

1. Functional Specifications

One of the more difficult processes in producing viable software is translating the design characteristics of the system into meaningful requirements for the programmers. Boehm, McClean, and Urfrig [2] vividly demonstrate the magnitude of the problem in their study of a large software project. The authors divided errors into two classes. These were design errors and coding errors. An error was considered a design error only if its correction caused a corresponding change to the design specifications. Of the total errors, 64 percent were design errors. This alone is enough to illustrate the need for a valid method of design specification. Even more disturbing was the time frame within the testing that the errors were discovered. Of the 54 percent that were not discovered until the acceptance, integration or delivery phases of testing, 45 percent were design errors. The remaining nine percent were coding errors. Errors discovered in these latter stages of system design would be much more difficult to correct than ones discovered early in the coding of the project. If the majority of errors are going to be of the design type and if these are going to be the more difficult type of errors to correct, it is necessary to have a good system of describing the design specifications to the programmers. The functional model is an attempt to build such a system.

When the functional model is used, the user should be required to define all functions of the system. The functional requirements would consist of a statement of the

activities of the system and the associated inputs and outputs. By requiring these functional requirements, the programmers are assured of having a complete detailed description of the system at the beginning of the project. This description should aid the programmers in their understanding of the overall requirements of the project and provide sufficent detail to properly code the project. This would reduce the number of design errors that would occur.

It is possible to over-specify the design of a system. This could prevent the programmer from choosing the most efficent method of coding the project. It could also introduce errors into the system, if the user does not have a in depth knowledge of computers. This trap is avoided using the functional requirements of the model. The details are presented in terms of functions of the system which is the area in which the user is most knowledgeable. The implementation of the functions is left to the programmer, who is in a better position to determine the proper method.

Another pitfall of system design may be avoided using the functional requirements. Frequently, test specifications are not available early in a project because testability is not considered to be a design parameter. Instead, test requirements are formulated as an afterthought when it is too late to influence the design [17]. Functional test specifications are defined as test specifications which are based on testing the stated functional requirements and observing the corresponding outputs of the system. Functional requirements should be incorporated in the functional test specifications. Detailed design should not commence until this information is available.

2. Documentation

The need for complete and usable documentation should be a primary concern with anyone involved with any phase of system design and testing. Poole [13] implies "that the lack of good documentation usually means that testing is not performed as thoroughly as it should be and debugging is that much more complicated." Another application of documentation is the maintainance of the system. Since the life cycle of a system is much longer than the development phase, the designers will probably not be available to help maintain the system. In addition, many different people may have had access to the software of the system. Any changes made by these people must be preserved for future use.

The functional model attempts to provide adequate documentation throughout the life cycle of the system. The concept is to force documentation to be an integral part of system development. Two documents have already been discussed. These are the functional requirements of the system and the functioal test specifications. These documents should form a segment of the documentation. These should be systematically updated as changes are made to the system.

The documentation should include other information as well. This could include a data base containing information about all errors that were found in the system during the life of the system to date. Unfortunately there is a tendency to ignore this aspect and to think of this type of information as something to discard once the error has been corrected [13].

Every incident must be recorded because an error that may appear insignificant to the user could be an important indicator once it is properly analyzed. The data

base could be used to classify modules that are the source of the majority of errors. This classification could be used to direct future testing and debugging. It could also be used to determine which modules are the most unreliable. This would provide a starting point if one desired to improve the reliability of the system. This would be particularly applicable if the module that is most critical to the system's operation is also the most unreliable. The data base could also be classified as to type of errors. This would be valuable information when designing a similar system.

Another form of documentation that should be incorporated into the plan for system testing is assertions. These are statements that are introduced into the code by the programmer. These state a fact about the design of the program. These statements may be treated as a comment card or used to produce code to check for the validity of the assertions. The appropriate action would be determined by a parameter passed to the complier. Two types of assertions could be employed within the model. The first would be global assertions. These would be in the form of specifications for intermodular actions of the system. An example of such an assertion would be

ASSERT RANGE OF ALL ARRAY INDICES IS 0 TO 100.

The other level of assertions would be local. The local assertions would be defined by the programmer but within the design specifications. An example of a local assertion would be

ASSERT RANGE OF I IS 10 TO 20.

These assertions would be a permanent feature in the

program. They could be activated on the local level to help test a module or on the global level to aid in introducing a change to the system. As such, these assertions would form an important part of the system documentation.

3. Test Inputs

Ideally, it would be proper to exhaustively test a system. This implies that every path in the logic of the program be executed and tested. Shooman [12] demonstrates that this will normally be impossible due to the large number of inputs required. The problem presented involved exhaustively testing an assembly language program which solved for the roots of a quadratic equation $Ax^2 + Bx + C = 0$. The computer was assumed to have a 12 bit word length and integer arithmetic was used. All syntactical errors had been elimated and all known special cases such as A = 0 and imaginary roots had been cared for. The input space to exhaustively execute this program involved 64×10^9 combinations of A, B, and C. The program had a run time of 240 microseconds per execution. The time to complete the entire execution of the program over the input space was 5,000 hours. To test the program, the solutions must be verified by some independent means such as a desk calculator or a different algorithm. This should be done in as many different ways as possible, since there is a probability that two independent solutions will compute the same wrong solution. Obviously, exhaustive testing is infeasible for even a small program.

The problem the tester must solve is how best to

select the subset of test inputs from the universe of possible inputs. A method for selecting the inputs for a test is to first identify and rank the modules in a system by the criticality of the modules to the mission success. It is seldom the case that all modules are equally valuable. A technique for determining criticality is to ascertain the consequences to the mission of a module malfunction. A malfunction in some modules would cause a mission abort, while others would result in a degraded mode of operation. The modules are ranked according to criticality. The time spent in testing each module can then be allocated using this ranking. The time allocation can be further refined by ranking the criticality of each sub-function of the module. This would be based on the criticality of the sub-function to the performance of the function by the module.

There are other factors that can be used to rank modules for testing purposes. One such criteria would be forcasted errors. Schneidewind [16] has developed a model of the occurrence of errors detected during functional testing of command and control software. Based on the results of a model that has characteristics similar to the one being tested, it would be possible to rank the modules in order of forecasted errors. Work is progressing in the area of developing relationships between program structure, program complexity and the ability to detect errors in a program [3]. Another method of obtaining such a ranking would be through the use of simulation. Critical modules could be identified by their high rate of failure in the simulation.

Once the amount of testing resources allocated to each module has been determined, the proper number of inputs for testing each module can be fixed. The problem then becomes one of selecting the best inputs to throughly test each module. The module represents a function which maps

the set of inputs into the set of outputs. The inverse correspondence could be used to obtain the set of inputs. Given this set of inputs, select test cases in order to cover the set as thoroughly as possible. Pay particular attention to the inputs that are involved in the control flow of the program. Once this has been done and if there are test resources still available, investigate unusual cases that may cause the system trouble. A possible source of unusual cases would be indicated by the set of inputs. Pick values that are combinations of the extremes of the ranges of inputs. Investigate combinations involving zeros. This is an excellent chance to investigate some of the "what if" questions.

4. System Representation

Having fully developed the notion of a module, it is necessary to investigate the method that will be used to represent a system as a collection of modules. A system is comprised of asynchronously operating application software modules, hardware modules and executives.

A system may be represented by the model. Fig 3 gives a generalized representation of a processing system. The system represented in this figure is comprised of two asynchronously operating executives, A and B. These are connected to two seperate control buses noted by Control Traffic Bus A and Control Traffic Bus B. Each bus connects the application software modules and hardware modules that are controlled by the executive on the bus. An example of the traffic on this bus is a hardware generated interrupt occurring at the conclusion of an input/output operation. A subsystem is comprised of one executive, the modules that it controls, and the control bus connecting the modules to the executive. There is a message traffic bus connecting all

modules. An example of the traffic on this bus is a module passing a computed value to another application module. External inputs and outputs are identified.

This representation of a system has many useful applications to testing. The model may be used to verify the correct functioning of two types of intermodule communications. The first is message traffic. The traffic on the message bus could be checked against the functional test specifications for correctness. The second concerns control traffic. The traffic on the control buses could be checked in a similar manner. Other problem areas that could be investigated using the model include:

* Are the various state transitions possible, based on the values of the resource states?
* Are there any blocked or deadlocked states?
* Are the amounts of time in each state excessive?
* When a module state transition occurs, are the resource state vectors correct?
* Are the times that a module holds resources excessive?

Fig 3 GENERALIZED REPRESENTATION OF A PROCESSING SYSTEM

III. SIMULATION

A. A SIMULATION OF THE MODEL

A simulation of the model was constructed. The simulation was an event store type of simulation. It was writtrn in FORTRAN to run on the Postgraduate School's IBM 360/67. The simulation used the model representation with the user providing a description of the system to be simulated. This description required the number of modules, the number of tasks, the precedence among tasks, number of resources and resource usage. A complete description of the simulation appears in Appendix A.

The simulation showed that the model could represent a system. A simulation of this nature could be useful in the test environment.

B. SIMULATION IN THE TEST ENVIRONMENT

1. Investigation Of Timing Problems

Timing problems are extremely hard to investigate in a real system due to the fact that any test equipment installed internal to the system disturbs the timing of the system. Such equipment installed external to the system may

not be able to gain the required information either because of synchronization problems or because of access problems. By using a simulation of the system, the tester may observe any timing parameter desired. The tester may even add timing parameters that are not found in the real system, but are necessary to completely evaluate some time critical operation. By doing this, the tester may observe timing problems that could not be observed on the real system. This addition could be made without disturbing the timing of the real system.

Another problem area of testing that could be investigated through the use of a simulated system is the reaction of the system to various rates of inputs. In the simulation of the model, it was possible to vary the mean time between arrival of inputs to the system. This parameter could be decreased on each run to determine the maximum input rate that the system could receive and still process an acceptable amount of the inputs. Another method would be to plot average time to process a complete input verus input rate. This graph could be used to determine an acceptable range of input rates for the system in order to completely process an input in a given time period. This method of anaylsis could be used when testing a system that has to produce outputs on a regular time basis, such as a system using a graphics display that has to be refreshed at a given rate. Another important use of this method of analysis would be to determine how much the input rate could be increased in the future.

If the time that a module spends in a particular state is expressed as a range instead of a constant, a simulation would be an invaluable aid to the tester in investigating the operation of the system. One approach would be to observe the operation of the simulated system with all modules functioning at the maximum time range.

Another method would be using various combinations of module operation times, selected from the ranges, to determine under what circumstances the system would fail or be in a degraded mode of operation. This method would give the tester the ability to check the system over all combinations of time ranges to prove the validity of the range and to identify critical areas within the ranges. This can easily be done on a simulated system but could be impossible to do on a real system because the tester would be unable to control the time the module spent in a state.

Another timing problem facing the tester is the timing of the system as a whole. Often the tester would like to slow the system down or perhaps speed it up in order to observe some particular action of the system. This would be important if the tester was unable to measure the output of a particular module because another output arrived before the first output could be measured. In a real system, it may be impossible to change the timing of each component of the system by the same amount. This would be particularly difficult in a multi-executive system. With a simulation of the system, the tester would be able to adjust the timing of the system in any manner he saw fit.

Some problems do not occur until the system has processed a large number of inputs. The tester may not be able to cycle the real system through such a large number of inputs due to time or some other constraint. However in a simulation, the time scale may usually be greatly compressed allowing the tester to cycle the system through many times. This would greatly increase the probabilty of discovering latent bugs.

2. Fault Insertion

Dijkstra [6] contends that "testing can only determine the presence of errors, not their absence." One approach to try to prove the absence of faults would be to know the reaction of the system to every possible error and combination of errors. Using this knowledge, one could simply observe the reaction of the system and state what errors were or were not present. Unfortunately, the set of every possible error, combination of errors and system reactions is an infinite set. Therefore the above approach would also fail to prove the absence of faults. However, the approach could be used to greatly expand the subset of errors that the tester could detect.

The tester may introduce a fault in the system. The reaction of the system to this fault could be catalogued for later reference. This information could be used to identify modules that are affected the most by a class of errors. This set of modules would be noted for special testing. This information could also be used to ensure the validity of the test plan. If the group of tests indicated in the test plan did not encompass the reactions observed in the simulation then the test plan would not be able to detect that particular fault.

3. Partial System Simulation

Frequently the tester will not have the time, assets, or motivation to perform a simulation of the entire system to be tested. In this situation, simulation of certain parts of the system may be desirable or the tester could choose to simulate the system in less detail. Campbell and Heffner [4] relate a case history illustrating this point. A simulation model was constructed of a system being developed. The skeleton system was working before the model was debugged. When the model was finally working, no one was

sure which version of the system the results were meant to represent. However, some of the designers used simple simulations that they developed to study certain aspects of the system. The authors concluded that "ambitious large-scale models generated by professional model makers are less helpful than simpler work done by the system developers themselves." Obviously a simulation with less detail was more useful in this case than a complete simulation.

Quite often in prototype testing, a module or module will not be present when the tests are scheduled to commence. This could be due to late delivery or to a module being modified after preliminary testing proved the module needed modification. This could also be caused by a planned action such as phased delivery. A simulation of this module would allow the tests for the rest of the system to continue. Simulating the missing module would be particularly easy if the system had been described in the form of a functional model. If all the information required to functionally represent the model is present then a simulation can be constructed from this information.

Another use of simulation involving less than the whole system is the test data generator. When the system is being tested in the laboratory environment, it may be necessary to simulate the inputs to a system. Since there is no reason to believe that all modules will be present during the entire test phase, the tester may desire to have the test data generator be able to simulate the output from any module. Thus the test data generator could also replace any missing module as the system was being tested. Not only would the generator have to act as a output generator but it must also act as a destination for outputs from other modules intended for the missing module. If these outputs were not accounted for, the reaction of the partial system

would not be truely representative of the reactions of the real system.

4. Pitfalls of Simulation

After having spent much time and effort to develop a simulation, the user may find that the simulation addressed the wrong problem or worse yet solved no problem at all. The validity of a simulation is the consistency between the simulation and the real system it represents. Proof of the validity of a simulation is almost impossible, especially if the real system has not yet been constructed. By the time the real system has been constructed and the validity of the simulation is disproved, irrevocable decisions may have been made based on test data from the simulation.

Although validity is a major problem in simulation, it is by no means the only problem. A list of problem areas that may cause a misunderstanding of the system being simulated is presented in Fishman [7]. These include input parameter misspecification, influence of initial conditions on data and misuse of estimates. Along with the list, there are suggestions on ways to control these problems.

Prototype testing may result in design changes. Each change requires a change in the model. If the tester has not allowed for such an ocurrence in budgeting his simulation resources, the model would represent a system totally unlike the real system. Also, there would be a time lag in modifying the model. This could have a serious effect on the test schedule if this contingency is not included in the plan.

IV. APPLICATION OF MODEL TO SYSTEM TEST

A. APPROACH

1. Test Plan

In order to apply the functional model to the problem of system test it is necessary to develop a test plan. A test plan should be created as part of the design plan. As a minimum the test plan should discuss the following major elements of the test:

* define modules
* define module states
* identify inputs and outputs for each module state
* identify module interfaces
* define resources and resource states
* identify resource usage for each module state.

The test plan must also include the functional requirements and functional test specifications. In addition it should include the test procedures. This would identify acceptance criteria, such as the allowable divergence between desired and actual output values, time duration of tests, allowable number and types of malfunctions, number and distribution of test replications, and means of checking test results. The

test plan should identify major testing milestones. These would identify major sections of testing that must be completed before system development can continue.

The test plan should document the subsystems that will be tested. This will require the development of a method of isolating a subset of the system to test it without the effects of the remaining system being introduced. These identified subsets of modules are called subsystems and will be used to test the system in stages.

Besides the modules in a subsystem, the test plan must also define a set of measurements which will indicate wheather correct outputs are being produced for given inputs and define the hardware and software locations of the measurements. The plan must describe how to instrument the system in order to obtain these measurements.

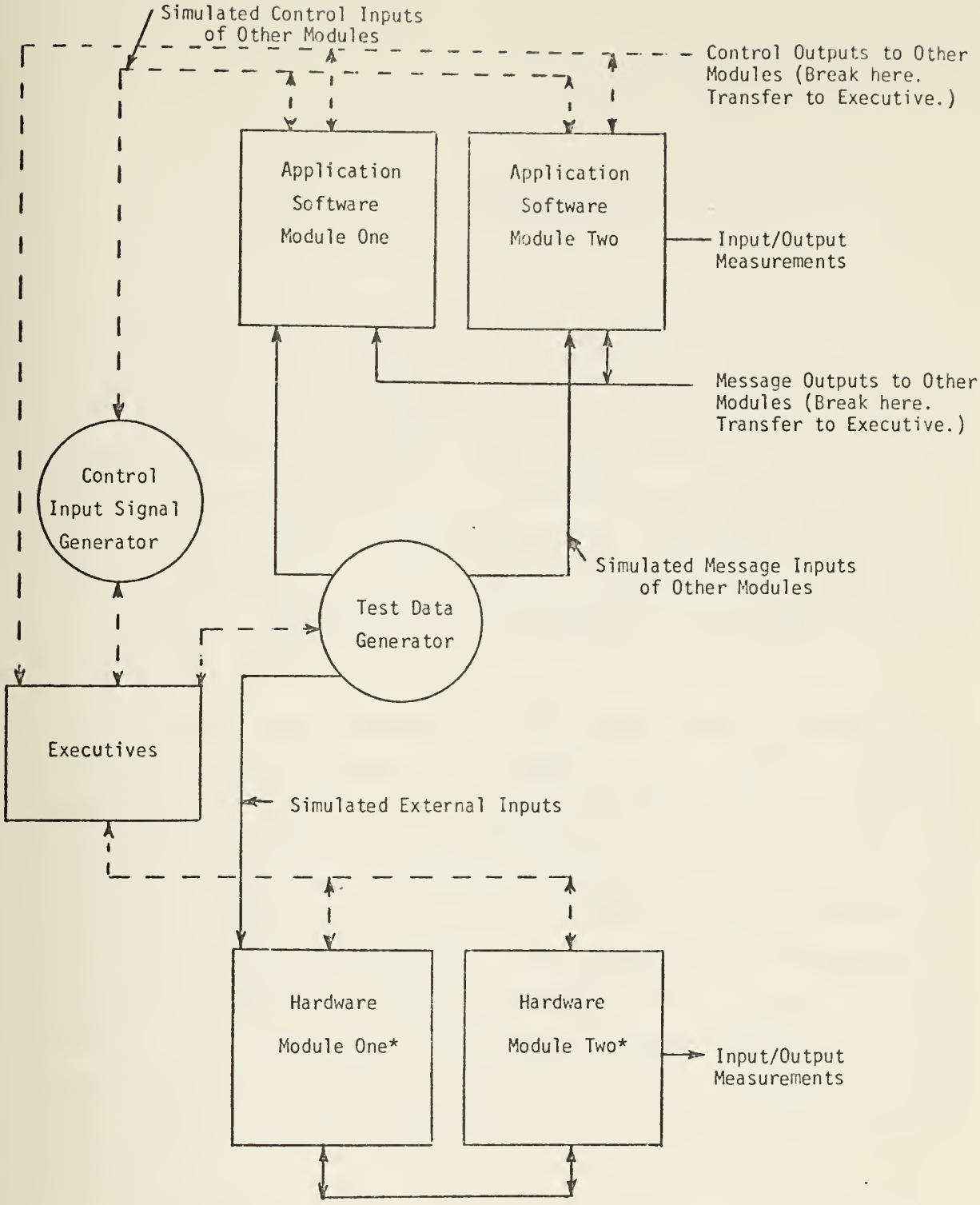
The test plan should develop some organizational structure. This would include who is to do the testing and the resources to be used in testing. The plan should include who is responsible for maintaining the documentation. This would include the test data, the error information, design changes and test modifications.

2. Subsystem Testing

The modularity of the model allows the user to commence testing as soon as possible. This will require the testing of a subsystem . The subsystem is defined in the test plan and testing will commence as soon as all the modules of the subsystem are available. This is illustrated in Fig 4. In this figure there are two application modules in the subsystem that are ready to be tested. The only hardware that is required for the functioning of the modules is Hardware Module One and Hardware Module Two. These are

connected to the application software modules by a test data generator. This is a program that will simulate external inputs and message inputs from missing modules. The set of external inputs and input sequence is based on typical operational scenarios and the critality of the various modules to mission success. another required program is the control input signal generator. This works in conjunction with the executive and generates simulated control inputs for the missing modules. In some cases this program could be developed to replace the executive itself during early testing, when the executive may not be available. The locations where input/output measurements are made are identified.

The system under test will be expanded as testing proceeds. When the next application subsystem becomes available, it and the hardware it uses could be added to the system. The subsystem could consist of a single module or a group of modules. The intent is to test the system in stages starting with the minimum number of modules, and increasing the number of modules as testing progresses, until the whole system is present.



*This is the only hardware required for functioning of application modules one and two.

Fig 4 SYSTEM TEST CONFIGURATION

V. CONCLUSIONS

There is a great need for better organized test methods. As the impact of computers on society increases, stringent requirements will be imposed on the reliability of computer systems. When missiles that can destroy a country are entrusted to the control of a computer, even one error is unacceptable. Recent developments for designing and testing computer hardware have done much to improve the reliability of systems. Unfortunately, development of software testing has not kept pace.

A model was presented that may provide a means for attacking the problem of testing a large software development project. The strength of the model is based on two concepts: modularity and functional representation. The benefits offered by a modular approach to testing have been discussed. Modularity appears to offer a viable solution to many of the testing problems. The functional representation concept is crucial to the development of a sound testing policy. The user must be given an opportunity to specify exactly what the system is expected to do. The tester must test against these specifications and not some artifical system.

A notation was provided for the model. The notation allows the tester to describe the attributes of the modules in a compact manner. The model was analyzed as a directed graph and the notion of a time domain of a module was introduced. Both of these concepts should prove useful in testing a system. The application of the model to testing

was explained. It should be emphasized that this system is not intended as a unalterable system that provides the solution to all testing problems. Rather, it is a beginning point for the development of a system test methodology that will be improved each time it is applied.

A simulation of the model was constructed and was described in the appendix. Uses of simulation in the test environment were discussed. Simulation has much to offer during the testing phase of system development. One of the difficulties encountered was construction of the simulation. There does not appear to be a good special purpose simulation language or system for testing. An interesting future development would be to expand the model. The system would permit the tester to describe the system to be developed. The system should contain a package of testing tools, such as debug and restart capabilities, that the user could invoke as needed.

APPENDIX A

DESCRIPTION OF SIMULATION OF MODEL

1. Definition of Terms

The following terms were defined in the text: state, module, task, and primary state. Their usage in this description of the simulation of the model will be consistant with these definitions. The status of a module was defined to be the information required to identify a particular module, and its state, that is executing a task for a given input. Thus, the status of a module identifies: module, state, task, and input number.

All times in the program were based on the standard time unit. The standard time unit may be defined as the user desires. The only requirement being that the user must be consistant throughout the program and that all times be integer numbers.

2. Program Characteristics

The simulation of the model was an event store type of simulation. In this type of simulation the time is advanced each time an event occurs instead of on a fixed interval. This prevented the occurrence of a large number of intervals during which there was no activity. This type of

simulation maintains the complete status of the system during the simulation because events are executed in the order in which they occur. The events in this simulation consisted of the arrival of inputs, the completion of a primary state, data collection events, and the termination event. The event that represented the completion of a primary state was always followed by the occurrence of the next primary state unless the input terminated or the resources required to enter the next primary state were not available. A data collection event was a time period during which the user could measure any parameter of the simulation. These occurred at regular intervals throughout the running of the simulation and could be used to gather data to compute such statistics as average queue size.

The programming language chosen for the simulation language was FORTRAN. The motivation behind this choice was to ensure the portability of the program. FORTRAN is not an ideal programming language for simulation. Other languages such as GPSS (General Purpose Simulation System) provide many of the building blocks needed for the simulation such as the clock and the calendar. Although FORTRAN does not provide these blocks, it is readily available at most institutions. Another limit to the portability of the program is the random number generator which was used in the simulation. The program used the LLRANDOM package available at the W. R. Church Computer Center. The random numbers were passed by subroutine calls. By adjusting these calls, a new random number generator may be introduced into the program.

The simulation was designed to be able to simulate a wide range of computer systems. This was implemented by allowing the user to input various characteristics of the system. These characteristics are discussed in the input section of the appendix. The resources used were identified

throughout the program only by number. The user was to assign both the type of resources the system uses and the amount of each resource available to the system. the only requirement was that all quantities were to be integer values. The workload for the system was defined as a system of tasks. The user was required to specify the priority of tasks. Modules were identified by number. The user was to specify which module or modules was to execute each task. The user was also to specify which resources and primary states were required in the performance of each task.

The simulation was programmed in a modular fashion. Each major function of the simulation consisted of a separate subroutine. This design was chosen to facilitate any modification of the program and to improve the comprehension of the program by another user. All major data structures and variables were made available to all subroutines by the use of common blocks.

3. Inputs

The simulation allowed the user to vary the input parameters listed below. The description is in the order in which the inputs were required by the program. If no range is given the parameter did not have limits placed upon it by the program. These limits were caused by the size of the data structure used in the program. The user could have chosen any reasonable value for the unlimited parameters. Subscripted variables indicate multidimensional data structures.

* MAXIN --- Determined the number of inputs the system may process at one time. Range was one to ten inputs.
* MEANIN --- Represented the mean interarrival time for the system. Units had to be standard time units.

* NUMRES --- Total number of resources required by the system for proper functioning of the system plus one additional resource. The first resource was used by the simulation as a time representation. Therefore, one additional resource was required. This allowed the user to specify the amount of time the resources were in use. The range of NUMRES was from two to ten.

* MAXRES(I) --- This vector gave the maximum amount of each resource available to the system. The units were unconstrained except time units had to be in standard time units. The range of I was from one to NUMRES.

* NUMTSK --- This was the number of tasks assigned to the system. The range was from one to ten tasks.

* PRECTSK(I,J) --- This matrix was used to give the precedence relationships among tasks. Each column was identified by a task. This task blocks all tasks listed in the column. The first element of each column was the number of tasks blocked by the given task. For example, if column number three contained the following numbers reading from top down: 2, 4, 6. This would indicate task number three blocked two tasks, namely task four and task six. The variable I had a range from one to NUMTSK. J was confined to the same range.

* MAXSEQ --- This was the maximum number of primary states that a module had to go through in order to execute a task. The range was one to eight states.

* DOTSK(I,J) --- This matrix showed the different sequences of primary states that a module was required to go through to complete one task. Each row represented the path for one module to perform one task. The first element of a row was the task, the second element was the module, and the remaining

elements were the sequence of primary states. A zero indicated the sequence had been completed. The variables I and J had a range of one to ten.

* ENDTR --- This was the number of different primary states for every module. The maximum value was thirty.

* TSKRES(I,J) --- This matrix gave the amount of each resource required for the execution of each primary state of a module and the associated task. Each row was the resources required by one task/state/module. Thus the first three elements of each row represented the respective task, state, and module associated with the resource requirements given in that row. The first entry represented the time that the resources would be required in that state. The variable I had a range from one to ENDTR. The range of J was from one to NUMRES plus three.

The following parameters were assigned values in the start subroutine of the program and may be varied as the user desires:

* MAXTIM --- This variable gave the total time the simulation was to be run. Units were standard time units. The value assigned was 1000000.

* DATINV --- This variable was the time between data collection events. Units were standard time units. The value assigned was 2500.The user was free to choose any value for this parameter. However, the value chosen should be realistic in terms of the other times used in the program.

* IX --- This variable was the seed for the random number generator. It could have had a value between 1 and 2147483647. A value of 47979951 was used.

This variable was designed as an input to the LLRANDOM random number generator.

System parameters were built into the simulation and were located throughout the program. These system parameters may be changed as the user sees fit but with greater difficulity than the ones listed above.

* Number of tables --- One system table was used in which the status of all modules awaiting resources was held until the resources needed for a particular state were available.

* Size of table --- The maximum number of module status entries that could be stored within the system queue. This was assigned a value of thirty.

* Service discipline --- The table entries were serviced in a first-in-first-out manner, based on time entered in the table.

* Resource allocation policy --- A module task was not executed until all resources for that module were available.

* Probability of a task --- Task branching was done on an equal probability basis.

4. Data Structures

The data structures which were used in the simulation were divided into four main classifications. These were the system table, the simulation calendar, the event records and the utility group.

The system table was represented by a group of vectors. A vector was a one dimensional array that was used

to store status information. The group of vectors were accessed by a common index. The index was stored on a separate vector in order of request for resources. Thus a complete description of any module waiting for resources could be obtained without a sequential search of the table. This facilitated the user specification of service discipline.

The simulation calendar was represented by a group of vectors in the same manner as the system queue. Besides the information describing the status of modules that were in service, this group also contained vectors representing simulation information such as event type and event time. The event time was the time representing the scheduled completion of the event. The event type differentiated among end of state events, data collection events, arrival of input events, and termination events. A zero in this vector represented an empty slot on the simulation calendar. Each time an event was selected the event time vector was searched sequentially to determine the next event occurrence time in the simulation.

The event records represent a diverse group of data structures used to preserve information about the events in the simulation. The general form of these data structures were matrices with each row representing one input that was being processed by the system and a column for each data element to be preserved for each state. The data elements were items such as time input entered system, time module used resources in a primary state, and the order in which primary states were executed. In order to avoid being overwritten, these matrices had to be printed out at the end of each sequence of tasks. The column index to these matrices were all based on the number of state transitions and each index was adjusted for the proper number of elements being preserved in the matrix.

Utility records were used to save information as it was being moved from the system queue to the simulation calendar.

5. Output

Two types of output were provided for in the simulation of the model. These were event reports and sequential reports. The event reports occurred whenever the event with which they were associated was terminated. Three types of event reports were used in the simulation. These were end of state report, end of task report, and termination report.

The end of state report occurred at the end of every primary state. The report contained the status of the module. The report gave timing information about the primary state just completed. This included the time the module was placed in the table, the time the module entered service, time the module finished service, total time in the system, total time in service, and total time in table. The report showed the amount of each resource available after the resources used in that state were returned to the system. A printout of the system table was shown after the next module status had been added to the table. All modules that were scheduled at that time were reported giving the status of each module. The report concluded with the status of all system resources after the scheduling activity had been completed.

The end of task sequence report occurred after the complete sequence of tasks for one input had been completed. The first part of the report gave the sequence of states, identifying each state by task and module. The second part of the report gave information about the timing of the task

sequence. This included the time the first module entered service, the time the last module finished processing, the total time required if the sequence of tasks had been processed serially, and the total time the input was in the system.

The termination report gave the total number of inputs that were lost by the system when it was already processing the maximum number of jobs allowed in the system. This report could be used to give averages from the data collection events such as average queue size and average percent of each resource being used by the system.

sequential reports were not used by the simulation but could have been initiated by the data collection event and used to give an instanteous picture of the system at regular intervals.

6. Program Flow

The program commenced with a call to subroutine start. This subroutine initialized all the variables and returned control to the main program. The next event was selected by the subroutine NEXT. This subroutine selected an event based on a sequential search of the simulation calendar. Control was returned to the main program which called the appropiate subroutine to handle the selected event.

If the next event was an arrival of an input, subroutine NEWINP was called. Subroutine NEWINP inserted the first module status on the system table with a call to the subroutine PUTONQ. this occurred only if there was space in the system for another input and if there was room in the system table. If there was no space in the system, the input

was counted as a lost input. If there was no room on the system table, the program was terminated with an error condition. With the exception of an error condition, the next arrival of an input was placed on the simulation calendar with a call to the subroutine ENTER. A call was then placed to the subroutine SKED. This subroutine checks to see if the current available resources of the system are such that a module on the system table may commence service. All modules were examined on a first-come-first-served basis and as many as possible were placed in service. The end of state event was then entered on the the simulation calendar. Control was then passed to the main program where the next event was again selected.

The subroutine ENDSTT was called. The resources used by the module were returned to the system. If the completion of the state did not complete the task, the next module status was placed on the table and subroutine SKED was called. If the state completed the task the next task was selected. The first module status of this task was then placed on the system table and again SKED was called. In either case STTDUMP was called to produce the end of state report. If the task was a terminating task, subroutine DUMPIN was called to produce an end of task sequence report. Control was then returned to the main program.

Two other types of events could occur. These were the data collection event and the termination event. A data collection event resulted in a call to subroutine WHAT. This subroutine collected the assigned information, scheduled the next data collection event, and returned control to the main program. The termination event occurred at the maximum simulation time and produced the termination report.

7. Desirable Features

The generality of the simulation and hence its applicability to any real system was restricted by building choices for parameters into the simulation instead of giving the user a choice. These parameters included the queuing discipline, resource allocation policy and the distribution of time spent in a state. In an ideal simulation of this nature, there would be many choices of these parameters, with the user having the ability to specify the discipline, policy, or distribution that best represented the system he intended to test.

During the development of the program, other features that would have added to the usefulness of this type of simulation were discovered. One such feature was a debug package. This could include such items as a stall alarm. This alarm would give a warning if a module was in the system table for an excessive time period. The length of this time period could be selected arbitrarily and then adjusted by the user based on the system description. This alarm could be used to detect modules that were deadlocked or to set a maximum time that a module could be in the table before a failure occurred. Beizer [1] indicated various other alarms that would appear useful in testing the simulation. These include watchdog alarms, cycle alarms, improper sequence alarms, and multiple or missed interrupt alarms (if the user chooses to install interrupts).

Another feature that would be useful in the testing environment would be a restart capability. The status of the system at the termination of the simulation could be saved in a dump file. The user would then be allowed to stop the system and make any observations he desired and then start from the same point. This ability would be particularly useful in investigating errors that occurred in a random fashion.

The above feature could be best utilized as an interactive program. The tester could watch the display of pertinent system data as the simulation operated. When an unusual occurrance was observed, the tester could stop the simulation and query parameters to determine what had taken place. Having made his observations, the tester would then have the choice of restarting the simulation, continuing from the point of interrupt, or saving the data produced on hard copy.

8. Limitations

The simulation that was programmed had some limitations for any practical use in testing. One of the major limitations was size. The number of tasks and other elements was limited. This was done to reduce turn-around time during the development of the program. Another major limitation was the amount of computer time required. The program could have been made to run faster; however, the computer time was affected when features were added or deleted after most of the simulation had been programmed.

LIST OF REFERENCES

1. Beizer, B., The Architecture and Engineering of Digital Computer Complexes, p. 641-665, Plenum Press, 1971.

2. Boehm, B. W., McClean, R. K., and Urfrig, D. B., " Some Experience with Automated Aids to the Design of Large-Scale Reliable Software", Proceeding International Conference on Reliable Software, p. 105-113, 1975.

3. Bradley, G. H., Green, T., Howard, G. T. and Schneidewind, N. F., " Structure and Error Detection in Computer Software", Proceedings AIIE National Conference, p. 6, 1975.

4. Campbell, D. J. and Heffner, W. J., " Measurement and Analysis of Large Operating Systems During System Development", Fall Joint Computer Conference, p. 903-914, vol. 33 part 1, 1968.

5. Dijkstra, E. W., " Notes on Structured Programming", Structured Programming, p. 1-82, Academic Press, 1972.

6. Dijkstra, E. W., " Structured Programming", Report of NATO Conference on Software engineering Techniques, p. 84-87, 1970.

7. Fishman, G. S., Concepts and Methods in Digital Simulation, p. 262-310, John Wiley and Sons, 1973.

8. Gruenberger, F., " Program Testing: The Historical

Prespective", Program Test Methods, p. 11-15, Prentice-hall, 1973.

9. Hetzel, W., " A Definitional Framework", Program Test Methods, p. 7-10, Prentice-hall, 1973.

10. Mills, H., " Top Down Programming in Large Systems", Debugging Techniques in Large Scale Systems, p. 41-53, Prentice-Hall, 1971.

11. Naval Postgraduate School, System Test Methodology, by Schneidewind, N. F., Bradley, G. H., Howard, G. T. and Montgomery, G. W., p. 39, January 1975.

12. The Polytechnic Institute of New York, Meaning of Exhaustive Software Testing, by Shooman, M. L., p. 10, January 2, 1974.

13. Poole, P. C., " Debugging and Testing", Advanced Cource on Software Engineering, p. 278-318, 1972.

14. The Rand Corporation, Software and Its Impact: A Quantitative Assessment, by Boehm, B. W., p. 51, December 1972.

15. The Rand Corporation, Some Information Processing Implications of Air Force Space Missions: 1970-1980, by Boehm, B. W., p. 45, January 1970.

16. Scheidewind, N. F., " Analysis of Error Processes in Computer Software", Proceedings International Conference on Reliable Software, p. 337-346, 1975.

17. Vyssotsky, V. A., " Common Sense in Designing Testable Software", Program Test Methods, p. 41-48, Prentice-hall, 1973.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Chairman, Code 72 Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	Professor N. F. Schneidewind, Code 55Ss (advisor) Naval Postgraduate School Monterey, California 93940	1
6.	LT. G. W. Montgomery, USN (student) Naval Air Development Center Warminster, Pennsylvania 18974	1